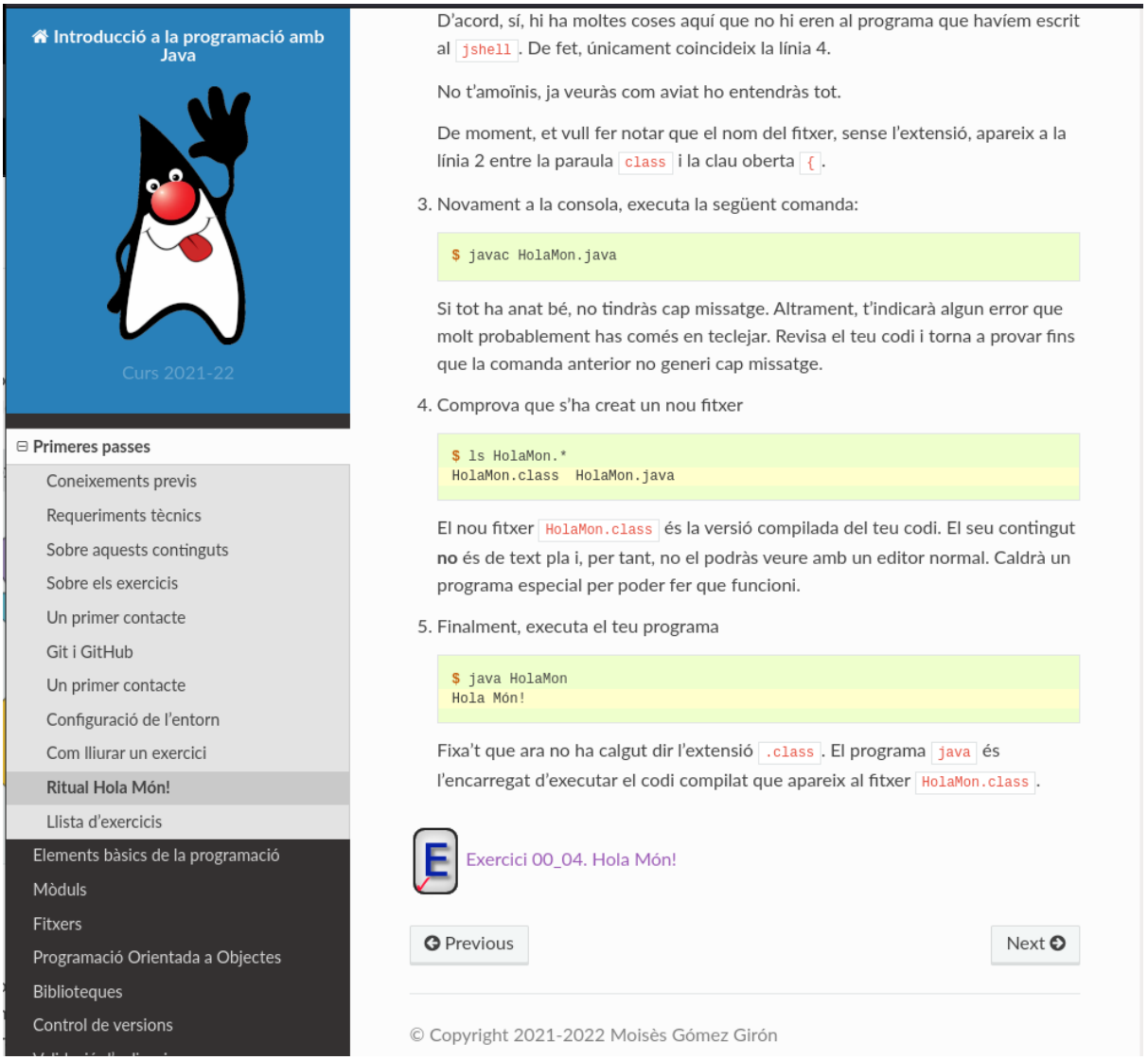
Learning with immediate feedback



Abstract

To teach students how to program, I have developed a number of tools aimed to shorten the time from delivery to feedback. At its best, it offers students, available any time, immediate rich feedback on their production, so they can deliver their assignments confidently.

This approach offers a number of additional benefits, such as, flexible schedule, unbiased evaluation, sense of fairness (no more score negotiation), improved engagement, more time during class to deal with diversity, clean comparison to perform statistical studies. Students get used to striving for strict fulfillment of requirements while dealing with current real professional tools and techniques.

These tools are integrated in a wider system that includes online contents specially tailored to students needs inspired in the flipped-classroom approach; a fair amount of meaningful (i.e. non abstract) exercises constructed in a way that each exercise includes a new feature

while recovering previously worked ones, to help integration; a positive and cheerful approach during class time encouraging participation, collaboration, assertive feedback, respect, professional orientation, and also very important, the idea that solving problems can be enjoyable and even fun.

In this article, I will focus on presenting the automation side.

Motivation

I am a computer engineer and a teacher of vocational training helping students to learn how to program for 18 years. My love for automating processes is compelled by the conviction that no human being should be forced to perform the same boring task more than twice when a machine can do it for us. To me, evaluating deliveries of the same exercise, again and again, is one of those boring tasks.

Back in my times as only-student, one of the most frustrating aspects of school was never knowing whether my work was right or wrong until the teacher delivered me their evaluation. These evaluations, from my point of view, systematically came late and often poorly described —sometimes just a raw value as 9 or A— So, most of the time this feedback-wannabe was useless at best.

That was not the *natural* way of learning. As you know, reality makes you aware you are not balancing properly on your bike, using very immediate —sometimes painful— feedback. Was that immediacy exclusive to learning body-control stuff?

When I became a computer developer, I discovered that when trying to run my flawed program, there was also immediate feedback delivered to me by tireless and merciless entities like compilers and interpreters. They would immediately tell me, for instance, that I forgot a semicolon or my program was trying to access a forbidden memory address.

Then, I became a teacher and I discovered myself delivering that same unuseful late and poorly described feedback to my students. Why? Of course, I was only a human with just so many hours in the day and, amongst many other things, so many deliveries to evaluate. Students did not complain, after all, my performance was just what they were used to. Nonetheless, I felt frustrated.

There must be something I can do, I thought. After all, deliveries are just information and that is what a developer is supposed to be specialized in dealing with!

So I decided to do something to fix, or at least, improve it. I decided to embrace the *way* of automation.

The *way* of automation

Embracing the *way* of automation is a continuous process. You don’t get there all of a sudden.

The first step toward shortening the feedback time is to require all the deliveries in **electronic format**. About twenty years ago, that was not that normal.

In the beginning, students would email me their deliveries deciding names, formats, subjects, etc. So you have me spending hours identifying the deliveries amongst hundreds of emails, then downloading the attachments and renaming them after the student's name. Here came the second requirement: **normalization**. It is great that students can express their originality (I love to see that variety of haircuts and colors), but they must conform to some requirements when it comes to a delivery: the subject of the email must be *exactly* "*delivery exercise_nr*", and the name of the file must be "*surname_name_exercisenr.zip"*. My first scripts (small computer programs) helped me to unzip all the deliveries in separate folders, conveniently named, so I could evaluate them uniformly.

Soon there came LMS (Learning Management System), Moodle in my case that put all the deliveries in the same place and allowed me to specify time constraints. Back then, there was no way to download all the deliveries as a pack, so I coded some new scripts to do so. In the same vein, other scripts would pick scores from a spreadsheet and place them on Moodle.

Once the mechanics of the deliveries were acceptably solved, the evaluation side got the focus. There are mainly two kinds of exercises depending on whether the product to be delivered is structured or unstructured.

Unstructured productions correspond to exercises like "Describe in your own words what is the function of *#include<stdio.h>* in your program". Sure you can code some scripts to check for document structure conformance (easier when normalizing document format) and other requirements like word count, but, in the end, I am still to find an acceptable way to escape from reading the contents if you want to give proper feedback. Reading is time-consuming so it is advisable to minimize this kind of exercise as much as possible. That is, **reduce manual evaluation**.

Structured productions are a whole other story. Let me define them in opposition to unstructured ones. A structured one is such a production that allows a program to fully evaluate it without requiring human intervention. The evaluation program will pick up the student's delivery and produce an evaluation result. This result can be as simple as a boolean pass/fail or as complex as a list of precisely-described missing, wrong or unexpected features.

The structured productions easier to come out, at least for me, were the computer programs. A computer program can be tested by running it with a given input and then comparing the produced output with the expected one.

Not only programs can be treated as structured productions. For example, a program can assess whether an XML document or a database conforms to the specified format and contents, or even a system ends in a certain state when the student performs the required actions using the operating system.

Am I hearing some teachers complaining in the form of " my subject is not in the computer development business; it requires mostly unstructured productions from my students."? If that is your situation, you know your alternatives for automation (mainly quizzes enriched with good feedback) Scripts can help you do some filtering for the textual productions, though and, who knows, as artificial intelligence advances, some of those productions could eventually become tractable as *structured*.

So, let us see what I have today.

Evil's in the details

This article gets a little bit more technical here, hopefully not too much as to annoy those of you less technical oriented, and not so little as to bore those techies amongst you.

The bunch of original scripts evolved into a sort of complex application that changes significantly every academic year. Wanting a better name, I call it *iesaval* (for *Institut d'Ensenyament Secundari* or Secondary Teaching Institute), and *avaluació* (for *evaluation*)

Currently, *iesaval* consists of around 60 scripts (over 20K lines of mostly Python code+tests) aimed to manage students, exercises, deliveries, tests, scores, reporting, and communication.

Six years ago, *iesaval* had a significant change. I started delivering part of the evaluation system to the students so they could get, finally, the immediate feedback I was yearning for all these years. This part of the system started as a set of Bash scripts and some JUnit. In the last editions, it became a more compact entity under the name of *prgtest*. Nowadays, *prgtest* is a Python script that is able to evaluate and provide rich feedback about each exercise, at the student's will. The current version is specialized to be run by the student as a command-line program on a standard GNU/Linux distribution (as Debian or Ubuntu) to evaluate their Java programs.

Another important improvement happened four years ago when Moodle was definitively replaced by GitHub, a sort of social media for developers that allows sharing code managed by arguably the most used control version system nowadays; that is *git*.

From then, the system gets improved every year. For example, two of this year's additions are the ability to deal with deadlines and performing auto-commits —i.e. *prgtest* does commit on behalf of the student to speed up interaction.

To deal with *iesaval*, students create a private git repository in GitHub and share it with me. Once registered the student and their repository, *iesaval* clones the student repository and copies the support files. Support files include the *prgtest* script, but also the *test specifications* for each exercise, and a *result report*.

The test specifications syntax is fairly flexible in the current version. It allows describing in a readable *Yaml* file for each exercise, the expected interaction with standard input/output, command-line arguments, files, and even databases, as well as specifying additional JUnit tests to check internals of the programs. The student can have a lot of freedom in their code and still get feedback about the deviation from the requirements.

The result report is an HTML document that describes the results of all the active exercises from *iesaval*'s point of view. It also includes information about deadlines and errors, if any, found.

The typical workflow for an exercise from the student's point of view is:

1. The teacher pushes the support files to the student repository

2. The student pulls the support files from their repository, reads the statement of the exercise and starts coding
3. Once the student is confident about their code, they run *prgtest* to get feedback about their work.
4. *prgtest* output guides the student on the next steps to take. For example, it can propose to compile the code or to commit changes to git. When an unexpected output is found, *prgtest* describes the failure. For example, it can specify the command used to run the student code, and when appropriate, the standard input or/and file contents provided to the code. Then it shows the expected output, the actual output of the program, and the difference. There are other more sophisticated tests that can show problems such as how a function failed to produce the expected results from the given arguments, or the wrong visibility of a class attribute.
5. Once the student considers the exercise is done, they can push it to the shared repository, from where *iesaval* will be able to retrieve it.
6. After some time —yes, this part is not immediate yet— the report is updated with the results from the delivery and the student gets it by pulling it from the remote repository.

From the iesaval point of view, the workflow is:

1. *iesaval* pulls the changes in the repositories of all the active students. This is usually started manually because I like to get a rough idea of which students have been delivering which exercises. But sometimes I schedule pulling, together with the rest of the steps, using standard GNU tooling (*cron*).
2. Once detected the exercises that have changed, *iesaval* runs *prgtest* on those delivered before the deadline.
3. Those passing *prgtest* will be quality tested. Quality tests are another set of tests apart from the ones performed by *prgtest*. These tests currently do not involve running again the student code and focus on other aspects of the delivered code. For instance, they can check whether the code contains headers, or that it does not include forbidden functionality as could be using the *break* clause within a *loop*. Failing here will generate error messages that will appear on the student's report.
4. With the results of basic and quality tests, *iesaval* updates the student's report
5. Finally, *iesaval* pushes the report to the shared repository so the student can get it.

The student can run *prgtest* as many times as they wish. Also, they can deliver as many times as wanted.

Let me do some math for you. To make things easy, suppose I have 50 exercises in a block with 60 students enrolled. Also, very unrealistically, let us assume I need a single minute to manually evaluate each exercise, and that the students just deliver each exercise once. With the given numbers it would take me around 50 hours to complete the evaluation! *iesaval* does all this within minutes while I am answering a question from one of my students. Furthermore, these assumptions are far from reality. Some blocks have way more than 50 exercises and for most of the exercises there are many subtle tests that, alone, would require minutes of manual evaluation. Also, students rarely deliver just once. Unfinished exercises and quality test failures often induce them to re-deliver; sometimes a significant number of times! Luckily I did not code the *complaint* module for *iesaval.* Therefore, it will

re-evaluate any change on the same exercise for the same student, again and again, until dead-line.

When a block of exercises has to be closed, *iesaval* offers some more help, for example, on extracting and placing data from spreadsheets and delivering final results by email. It will also help me with a number of small management utilities like checking remotes are private or integrating exercise statements within the course contents.

An important reminder: if you plan to implement something like *iesava*l: remember that running arbitrary code on your machine can bring you undesired outcomes. Of course, it is not easy that a student dares trying to break into your machine, taking into account that all the revealing traces stored in *git*, *GitHub*, etc, would focus directly on them. Nonetheless, **it is possible** to have your system compromised. *iesaval* has implemented some protections and I definitely encourage you to do the same.

Pros & cons

As with everything in life, a system such as iesaval offers a number of benefits but also has its drawbacks.

Starting from the positive points, maybe the rock star is the possibility to offer **immediate feedback** to the student, but it is not the only one I have found.

Closed in appreciation is the possibility to evaluate students without bias. With **no biased evaluation,** I mean that it is not required for the teacher to perform any effort on blinding the evaluation so as not to be influenced by possible prejudices or preferences that could discriminate against the students by gender, race, age, personal sympathies/antipathies, etc. Of course, most of us want to believe we are immune to all these flaws but, as any teacher knows, even taking precautions like blind-evaluating, it is not the same to evaluate a mediocre exercise after a sequence of bright ones, as evaluating the very same mediocre exercise after a sequence of really poor ones.

The always-available prgtest offers the students high **flexibility to schedule** their work. My typical student is an adult person, often with familiar and/or job loads. The possibility to work on their assignments at their best moment is an important feature.

The students tend to perceive a **sense of fairness** in the evaluation. You never get the typical “why Anna’s got an A while I get only B?”. There is **no score negotiation** either. If *iesaval* is doing wrong for someone, it is doing wrong for everyone.

Students, thanks to the immediate feedback, tend to **improve engagement** in their studies. Some have even reported that they feel like they are playing to defeat *prgtest* as if they were in a game. This was to me an unintended but highly welcomed outcome, as you can imagine.

Having a tireless *prgtest* that will tell the students how they are doing with their exercise, also frees the teacher from a lot of questions so they can dedicate **more time to dealing with diversity**. That is, to devote more time to the students with more difficulties and, what is still better, to enhance the experience of those bright students that normally get bored without attention because others less bright consume all the teacher’s time.

The no-bias evaluation property brings us another very nice benefit, the possibility to cleanly **compare students** in different groups and academic courses. While this is a very promising aspect, I have not yet explored it extensively. However, there are some first intuitions that can be validated by collected data, since it consistently seems to agree that the average age and the cardinality of the group do have an impact on the average performance of the group. You might argue that everybody knows that more mature and less crowded groups tend to perform better. I will say the same but now I can offer some juicy data to back my assertion. While the amount of students I have, might not be enough to make these stats significant, I consider it definitely better than not having them. I am even playing with the idea of feeding some machine learning algorithm to help me detect potential learning problems or cheating, and even suggesting exercises to reinforce weakly learned concepts.

Finally, an interesting advantage of this approach for developers is that they get used to fulfilling requirements strictly, dealing with automated testing, version management systems and GNU/Linux for free. All these are valuable skills for their future profession.

For the evilest side of *iesaval*, allow me to devote the next section.

Playing cat and mouse

While iesaval is nowadays a fairly robust piece of code, significantly protected by a bunch of unit testing it keeps revealing new weaknesses as I use it. Over the years, students have aptly proven over and over their creativity in finding these weaknesses.

I like to think about these weaknesses as challenges. They are both.

Let me give you some of the *many many many* examples I have found, fought and still fight, over the years.

The first one appeared on a very initial version of *iesaval*. The scripts were still coded in Bash script and the deliveries came from Moodle. Suddenly the evaluation stopped working with a cryptic message, and it took me some weeks to debug it. At the end, it was something very simple. A student managed to deliver a file with a carriage return within its name and that broke the system. The solution required some research and a small tweak so the evaluator script would read file names as a raw string.

Now, a more recent example. Some exercises require the student to give a certain answer depending on the input. The way tests work right now requires a finite number of well-defined tests. Some students discovered that instead of programming the requirements, *prgtest* would pass a code consisting of a list of conditionals in the form “if the input is A then result is a, else if the input is B then the result is b…”. Every time *prgtest* would complain, it would provide for new input and its expected output that should be added to the code. Furthermore, since I’m currently not performing extra running tests, they found that there was only a small chance to get caught if the exercise was finally evaluated manually. Clever if you are a lazy student, eh? To cope with this situation, I defined a number of solutions and finally implemented the cheapest one: a quality test that checks if the input is not present in the code.

While up so far the majority of the students have engaged with the *prgtest* way, there is always a number of them that do not. The main complaints are excessive strictness and too much work. Some students manifest to feel uneasy when they know they will be evaluated by a program instead of a human. Fortunately, as the system evolves, the number of these complaints have decreased significatively and, after properly explained, most of the complaining students understand that getting human evaluation means giving up some interesting features like immediate feedback.

Excessive workload is also a very common complaint, and not a particularly easy one to tackle. For some students, there are too many similar and basic exercises while for others, these same exercises are seen as too steepy and would appreciate some more redundant exercises for further practice. My most promising bet to deal with this problem is **personalization**. That is, being able to request different exercises depending on the student's previous performance. This is one of the main lines of research for the future *iesaval*.

The student's willingness to share their results represents a problem of its own. Every year I have to modify, at least slightly, different aspects of the exercises. Otherwise the students can find the solutions somewhere. I'm fighting this from different angles. To begin with, I try to explain to the students that they and the future students will not get the most from the experience if they find working solutions for the exercises. There are also a number of scripts aimed to help me in the construction and modification of exercises. Finally, the traces they leave when working on an exercise can be analyzed and some "cheating" patterns can be easily detected. Anyways, I feel that these solutions are partial and this problem constitutes another important line of research.

With all these in mind, what is to be expected from the future *iesaval*?

Wrapping up

I have presented *iesaval*, a work-in-progress set of tools that allows immediate feedback on students' productions. While *iesaval* offers multiple advantages, there are also important lines of research to improve it.

One might think that being able to evaluate effortlessly as many deliveries as required, should leave the teacher without anything to do other than picking up a nice beverage and relaxingly observing from the reporting tools how the students learn without supervision. Quite the opposite! This kind of approach is highly **time-consuming**. And the cat-and-mice game is just part of the story.

The number of improvements in my backlog is overwhelmingly high. Ranging from fixing detected bugs, visual improvements, cheat detection enhancement and even personalized exercise suggestions based on the student's learning pattern. And there is still more! This article has not dealt with the online contents (i.e. the study materials). During the last years, I have experimented with story-telling techniques, and they are producing quite significant results. For example, one of the characters that appears in the contents is a cat (yes, a cat!) named Renat. Traditionally some students bring to the class jokes, drawings and even poems! This year, a group of last year's students presented a project to a contest and the name of the team was…? Any guesses? It was *Cat Renat*!

I firmly believe that the combination of immediate —or at least fast— feedback with meaningful exercises and contents in a respectful and playful environment, creates a memorable learning experience for my students.

I have already said that this approach is highly time-consuming. What I have not yet told you is that it is also **highly satisfying**. What else could a teacher wish other than seeing their work is both useful and fun?

If you ask me what single event could significantly improve *iesaval*, with no doubt I would answer that some passionate teacher-plus-developer joins me in this adventure. Maybe you?<